Linked List

Introduction

- List of elements which are connected in sequence to each other by a set of pointers.
- Commonly used linear data structure.
- Each element is known as a node.
- A node consists of two parts
  - Data (value or values to be stored in a node).
  - Pointer (links or references to other nodes in a list).
- Types
  - Singly, Doubly, and Circular.

Contd…

- Advantages
  - Dynamic in nature, i.e. allocates memory when required.
  - Insertion and deletion operations can be executed easily.
  - Stacks and queues can be implemented easily.
  - Reduces the access time.
  - Efficient memory utilization, i.e no need to pre-allocate memory.
- Disadvantages
  - Wastage of memory as pointers require extra memory space.
  - No random access; everything sequential.
  - Reverse traversal is difficult.
  - Memory space restriction as new node can only be created if space is available in heap.

Operations

- Traversal (Searching, Displaying)
- Insertion
  - At the beginning.
  - At the end.
  - At a specific location.
- Deletion
  - At the beginning.
  - At the end.
  - At a specific location.

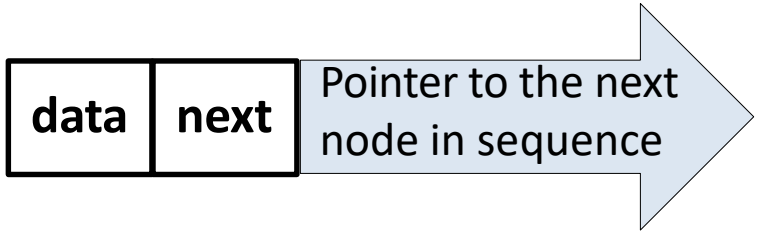
Singly Linked List

Introduction

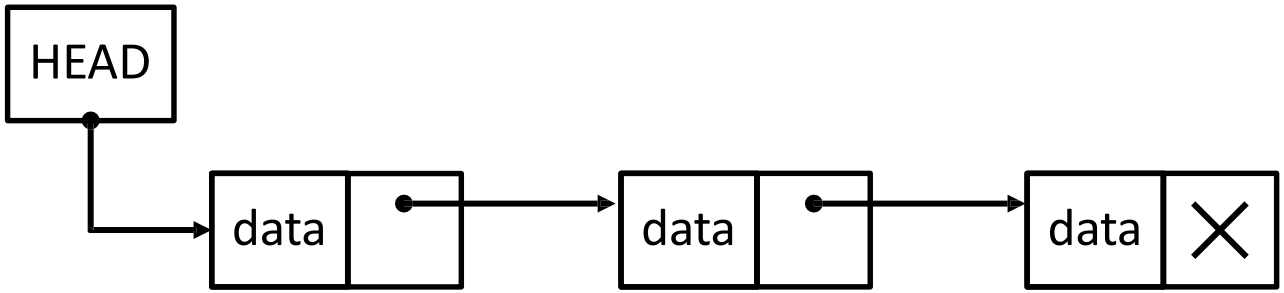
- The most basic type of linked list.
- Two successive nodes are linked together as each node contains address of the next node to be followed, i.e. successor.
- A node may has multiple data fields but only single link for the next node.
- Only forward sequential access is possible (or unidirectional).
- Address of the first node is always stored in a reference node known as **front** or **head**.
- The last node does not have any successor and has reference to **NULL**.

Contd…

- Pictorial representation of a node



- Pictorial representation of a singly linked list



Creation

```
struct node
{   int data;
    struct node *next; };
```

- Define node structure.
- Declare a NULL initialized head node pointer to create an empty list.

```
struct node *head = NULL;
```

- Dynamically allocate memory for a node and initialize all members of a node.

```
struct node *temp =
                    (struct node *) malloc (sizeof(struct node));
int num;
scanf("%d",&num);
temp -> data = num;
temp -> next = NULL;
```

Create SLL (Implementation in C++)

```
class Node {
public:
    int data;
    Node* next;

    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class LinkedList {
private:
    Node* head;

public:
    LinkedList() {
        head = nullptr;
    }
```

```
void main()
{
 LinkedList myList;
}
```

Contd…

```
head = temp;
```

- Link the new node temp in the existing empty list.
- Again dynamically allocate memory for a node and initialize all members of a node.

```
*temp = (struct node *) malloc (sizeof(struct node));
scanf("%d",&num);
temp -> data = num;
temp -> next = NULL;
```

- Link the new node temp in the existing list at head.

```
temp -> next = head;
head = temp;
```

- This process is repeated for all the nodes. A node can be inserted anywhere in the list.

Search an element in the list

- **Algorithm** search(head, num)
- **Input**: Pointer to the first node (**head**) and a value to search (**num**).
- **Output**: Appropriate message will be displayed.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Return.
4. Initialize a node pointer (**temp**) with **head**.
5. while (**temp** is not **NULL** AND **temp**[**data**] is not equal to **value**)
6. **temp** = **temp**[**next**]
7. if (**temp** is **NULL**)
8. Print [**Element not found**].
9. Else
10. Print [**Element found**].

Search an element in the list

```
// Function to search for an element in the linked list
    bool searchElement(int key) {
        Node* current = head;
        while (current != nullptr) {
            if (current->data == key) {
                return true;
            }
            current = current->next;
        }
        return false;
    }
};
```

```
// Test search function
void main()
{
LinkedList linkedList;
 int key = 3;
    if (linkedList.searchElement(key))
{
        std::cout << "Element " << key << " is found in the linked list." << std::endl;
    }
 else
{
        std::cout << "Element " << key << " is not found in the linked list." << std::endl;
    }
}
```

Display elements in the list

- **Algorithm** display(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: Display all the elements present in the list.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Return.
4. Initialize a node pointer (**temp**) with **head**.
5. while (**temp** is not **NULL**)
6. Print [**temp**[**data**]].
7. **temp** = **temp**[**next**].

Display elements in the list

```
// Function to display the elements in the linked list

void display()
{
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }
```

Insertion at beginning of the list

- **Algorithm** insertBeg(head, num)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output**: Node with value **num** gets inserted at the first position.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**.
3. **temp**[**next**] = **head**.
4. **head** = **temp**.

Insertion at beginning of the list

```
 void insertAtBeginning(int value)
{
      Node* newNode = new Node(value);
      newNode->next = head;
      head = newNode;
   }
```

Insertion at end of the list

- **Algorithm** insertEnd(head, num)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**).
- **Output**: Node with value **num** gets inserted at the last position.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**
3. **temp**[**next**] = **NULL**
4. If (**head** == **NULL**)
5. **head** = **temp**
6. Else
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1**[**next**] is not equal to **NULL**)
9. **temp1** = **temp1**[**next**]
10. **temp1**[**next**] = **temp**

Insertion at end of the list

```
void insertAtEnd(int value)
{
      Node* newNode = new Node(value);
      if (head == nullptr) {
         head = newNode;
      } else {
         Node* current = head;
         while (current->next != nullptr) {
            current = current->next;
         }
         current->next = newNode;
      }
   }
```

Insertion after a specific value in the list

- **Algorithm** insert(head, num, value)
- **Input**: Pointer to the first node (**head**) and a new value to insert (**num**) after an existing **value**.
- **Output**: Node with value **num** gets inserted after node with **value**.

1. Create a node pointer (**temp**).
2. **temp**[**data**] = **num**
3. **temp**[**next**] = **NULL**
4. If (**head** == **NULL**)
5. **head** = **temp**
6. Else
7. Initialize a node pointer (**temp1**) with **head**.
8. while (**temp1** != **NULL** AND **temp1**[**data**] != **value**)
9. **temp1** = **temp1**[**next**]

Contd…

10. if (**temp1** == **NULL**)
11. print [**Node is not present in the list**]
12. else if (**temp1**[**next**] is **NULL**)
13. **temp1**[**next**] = **temp**
14. else
15. **temp**[**next**] = **temp1**[**next**]
16. **temp1**[**next**] = **temp**
17. end if (line 10).
18. End if (line 4).

Insertion after a specific value in the list

```
void insertAfterValue(int existingValue, int newValue) {
    Node* newNode = new Node(newValue);
    Node* current = head;

    // Special case: empty list
    if (head == nullptr) {
        head = newNode;
        return;
    }
```

```
 // Traverse the linked list to find the existing value
    while (current != nullptr && current->data != existingValue) {
        current = current->next;
    }

    if (current == nullptr) {
        std::cout << "Value not found. Node not inserted." << std::endl;
    } else {
        newNode->next = current->next;
        current->next = newNode;
    }
}
```

Insertion after a specific position in the list

```
 void insertAtPosition(int value, int position)
{
      if (position < 0) {
         std::cout << "Invalid position." <<
std::endl;
         return;
      }

      Node* newNode = new Node(value);

      if (position == 0) {
         newNode->next = head;
         head = newNode;
      } else {
         Node* current = head;
         int currentPosition = 0;
```

```
while (current != nullptr && currentPosition <
position - 1) {
            current = current->next;
            currentPosition++;
         }

         if (current == nullptr) {
            std::cout << "Position out of range."
<< std::endl;
            return;
         }

         newNode->next = current->next;
         current->next = newNode;
      }
   }
```

Delete from beginning of the list

- **Algorithm** deleteBeg(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: The first node gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. **head = head**[**next**]
6. Release the memory location pointed by **temp**.
7. end if

Delete from beginning of the list

```
void deleteAtBeginning()
{
      if (head == nullptr) {
         std::cout << "Linked list is empty.
Nothing to delete." << std::endl;
      } else {
         Node* temp = head;
         head = head->next;
         delete temp;
      }
   }
```

Delete from end of the list

- **Algorithm** deleteEnd(head)
- **Input**: Pointer to the first node (**head**).
- **Output**: The last node gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. while (**temp**[**next**] is not **NULL**)
6. initialize a node pointer (**pre**) with **temp**.
7. **temp** = **temp[next]**
8. if (**temp** == **head**)
9. **head** = **NULL**
10. else
11. **pre**[**next**] = **NULL**
12. Release the memory location pointed by **temp**.
13. end if

Delete from end of the list

```
 void deleteAtEnd()
{
      if (head == nullptr) {
         std::cout << "Linked list is empty.
Nothing to delete." << std::endl;
      } else if (head->next == nullptr) {
         delete head;
         head = nullptr;
      } else {
         Node* current = head;
         while (current->next->next != nullptr) {
            current = current->next;
         }
         delete current->next;
         current->next = nullptr;
      }
   }
```

Delete a specific node from the list

- **Algorithm** deleteSpecific(head,num)
- **Input**: Pointer to the first node (**head**) and a value **num** to be deleted.
- **Output**: The node with value **num** gets deleted.

1. If (**head** == **NULL**)
2. Print [**List is Empty**].
3. Else
4. initialize a node pointer (**temp**) with **head**.
5. while (**temp** is not **NULL** AND **temp**[**data**] is not equal to **value**)
6. initialize a node pointer (**pre**) with **temp**.
7. **temp = temp[next]**
8. if (**temp** is **NULL**)
9. Print [**Element not found**].
10. Return.

Contd…

12. else if (**temp == head**)
13. **head** = **head[next]**
14. else if (**temp**[**next**] **== NULL**)
15. **pre**[**next**] = **NULL**
16. else
17. **pre**[**next**] = **temp**[**next**]
18. Release the memory location pointed by **temp**.
19. end if (line 8).
20. end if (line 1).

Delete a specific node from the list(C++)

```
void deleteNodeWithValue(int value) {
        if (head == nullptr) {
            std::cout << "List is Empty." << std::endl;
            return;
        }

        Node* temp = head;
        Node* pre = nullptr;

 // Traverse the linked list to find the node with the
specified value
        while (temp != nullptr && temp->data != value)
{
            pre = temp;
            temp = temp->next;
        }
```

```
if (temp == nullptr) {
            std::cout << "Element not found." <
std::endl;
            return;
        }

        if (temp == head) {
            head = head->next;
        } else if (temp->next == nullptr) {
            pre->next = nullptr;
        } else {
            pre->next = temp->next;
        }

        delete temp;
    }
```

Delete a specific positon from the list

```
void deleteAtPosition(int position)
{
    if (position < 0) {
        std::cout << "Invalid position." << std::endl;
        return;
    }

    if (head == nullptr) {
        std::cout << "Linked list is empty. Nothing to delete." << std::endl;
        return;
    }
if (position == 0) {
        Node* temp = head;
        head = head->next;
        delete temp;
    } else {
        Node* current = head;
        int currentPosition = 0;
```

```
while (current != nullptr && currentPosition < position - 1) {
            current = current->next;
            currentPosition++;
        }

        if (current == nullptr || current->next == nullptr) {
            std::cout << "Position out of range." << std::endl;
            return;
        }

        Node* temp = current->next;
        current->next = current->next->next;
        delete temp;
    }
}
```